case for the downgrading procedure illustrated in Figure 7. Here, the input and output Ţles are hard-coded. (Note that a more general downgrading function might permit the input and output Ţles to be function parameters, in which case the results would parallel those of the guard.) Writing from a high input to a low output violates the type inference rules and no type results, as indicated by (?). As in the case of the guard, the editor has identiŢed code for which further analysis is needed. Ideally, such code should be isolated in its own domain to be executed by multilevel subjects, while the remainder of the application can be executed by single-level subjects.

4. Summary

In this paper, we have presented ongoing work to develop a practical tool to assist the developers of trusted applications. Our tool does not take the place of the careful design and analysis that should be applied to the development of TCB software, however, it does permit software developers to isolate code which violates policy from that which is benign. We anticipate that the editor could be employed in the development of very large software applications and that, as is the case with all other tools used in the development of trusted systems, the editor would be maintained under a life-cycle assurance program commensurate with the target evaluation class of the intended trusted application. Software certiŢed by the editor provides an additional level of conŢdence that the security policy will not be violated. This can be especially important in environments where the same code is executed by both single-level and multilevel subjects.

Currently, the editor implements only that part of the type system that guarantees programs do not violate *explicit* information ţow policy . We plan to extend the system to handle what Denning calls *implicit* information ţows as well. This will address *legitimate* channels by which processes can transmit information between security classes [12]. Some progress has been made in this regard. However, covert channels will not be considered. Finally, in some cases, the system may claim there is a violation when really there is not. This is also true of Denning's system [6] and is a consequence of the unsolvability of deciding security.

References

[1] *Department of Defense Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, National Computer Security Center, June 1985

[2] Bell, D.E. and LaPadula, L.J., *Secure Computer Systems: Mathematical Foundations*, Vol. I-III, ESD-TR-73-278, The MITRE Corp., Bedford Mass.

[3] Brix, H. and Dietl, A., "Formal Construction of Provably Secure Systems With Cartesiana" *Proc. 1990 IEEE Symp.on Security and Privacy*, Oakland, CA, May 1990, pp. 319-331.

[4] Denning, D., *Secure Information Flow in Computer Systems*, Ph.D. thesis, Purdue University, May 1975.

[5] Denning, D., "A Lattice Model of Secure Information Flow," *Communications of the ACM*, 19, 5, 1976, pp. 236-242.

[6] Denning, D. and Denning, P., "CertiŢcation of Programs for Secure Information Flow," *Communications of the ACM*, 20, 7, 1977, pp. 504-513.

[7] Denning, D., "Cryptographic Checksums for Multilevel Database Security," *Proc. 1984 IEEE Symposium on Security and Privacy*, Oakland, CA, April 1984, pp. 52-61.

[8] Denning, D., *Private communication*, March 1995.

[9] Fuh, Y.C. and Mishra, P., "Type Inference with Subtypes," *Theoretical Computer Science*, **73**, 1990, pp. 155-175.

[10] Gordon, M., "HOL, A Proof Generating System for Higher-Order Logic," in *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. Subrahmanyam, (Eds.), 1988, pp. 73-128.

[11] Irvine, C. E., "A Multilevel File System for High Assurance," *Proc. 1995 IEEE Symp.on Security and Privacy*, Oakland, CA, May 1995, pp. 78-87.

[12] Lampson, B., A Note on the ConŢnement Problem, *Communications of the ACM*, 16, 10, 1973, pp. 613-615.

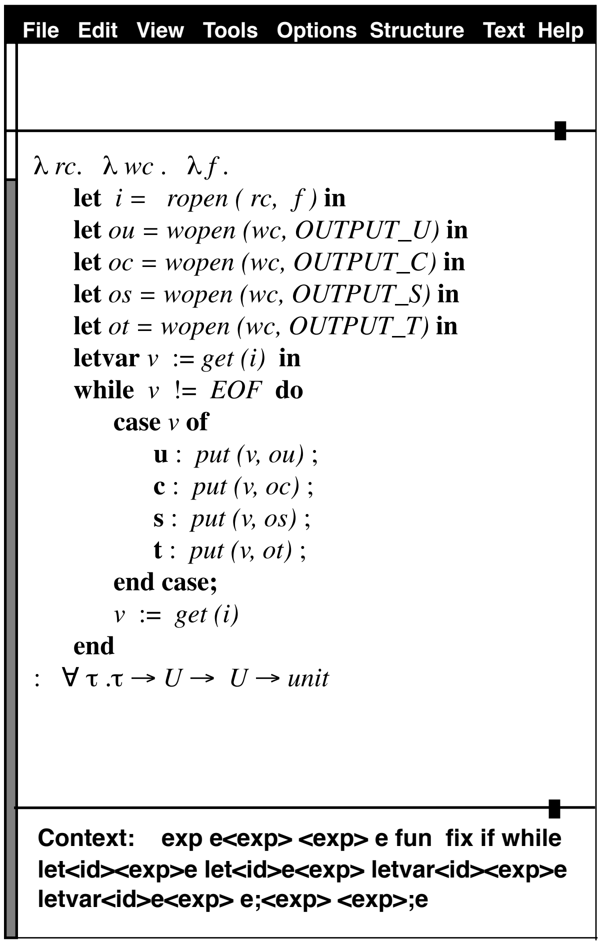
[13] Schell, R. R., and Brinkley, D., "Concepts and Terminology for Computer Security," in *Information Security: an Integrated Series of Essays*, M. Abrams, S. Jajodia, and H. Podell (Eds.), IEEE Computer Society Press, Los Alamitos, CA, 1995, pp. 40-97.

[14] Smith, G. S., *Polymorphic Type Inference for Languages with Overloading and Subtyping,* Ph.D. Thesis, Department of Computer Science, Cornell University, Technical Report 91-1230, 1991.

[15] Stansifer, R., "Type Inference with Subtypes," *Proc. 15th ACM Symposium on Principles of Programming Languages*, 1988, pp. 88-97.

doing something undesirable without users ever knowing it. When our system is used to develop code, however, it alerts one to rogue programs like this one.

3.0.2 A guard: Observe that in the rogue Ţle copy example, the editor did not detect a policy violation in the procedure. It merely concluded that the rogue procedure can be used to copy only unclassiŢed Ţles if policy is not to be violated. To rid ourselves of this restriction, we can simply delete the offending put statement to restore the generality of Ţle copy. In some cases, generality cannot be restored so easily.



```
λ rc.  λ wc .  λ f .
    let  i =   ropen ( rc,  f ) in
    let ou = wopen (wc, OUTPUT_U) in
    let oc = wopen (wc, OUTPUT_C) in
    let os = wopen (wc, OUTPUT_S) in
    let ot = wopen (wc, OUTPUT_T) in
    letvar v  := get (i)  in
    while  v  !=  EOF  do
        case v of
            u :  put (v, ou) ;
            c :  put (v, oc) ;
            s :  put (v, os) ;
            t :  put (v, ot) ;
        end case;
        v  :=  get (i)
    end
:  ∀ τ .τ → U →  U → unit
```

```
Context:    exp e<exp> <exp> e fun  fix if while
let<id><exp>e let<id>e<exp> letvar<id><exp>e
letvar<id>e<exp> e;<exp> <exp>;e
```

Figure 6. Guard procedure

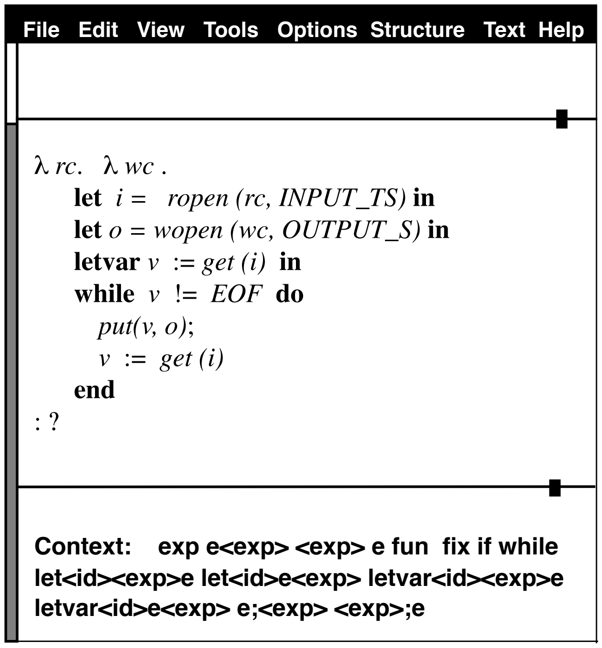
As an example, consider a guard. Here we assume the precise deŢnition of a guard as described by Schell, reported by Denning [7], and described by Schell and Brinkley [13]. It has to partition the elements of a multilevel stream according to their security classes. Its operations are based upon credible access labels which are attributes of the stream elements and may be certiŢed to be in compliance with security policy. A guard is deŢned in Figure 6. Here *f* is a multilevel stream and the Ţles opened for reading and writing have the classiŢcations suggested by their names. The type inferred for the guard implies that the guard can be called only with a single-level input stream consisting exclusively of unclassiŢed values:

$$\forall \tau.\, \tau \rightarrow U \rightarrow U \rightarrow \mathit{unit}$$

But a guard cannot be limited this way, so in order to recover its full generality, it must be developed outside of the editor and veriŢed some other way. The editor is useful in such situations because it helps to identify system components where more complex forms of analysis are needed. This greatly reduces the amount of code that has to be veriŢed by more complex techniques. In our system, the guard would have to be separately veriŢed outside the editor and re-introduced into the system by supplying an appropriate typing for it, say,

$$\forall \tau, \gamma \ \textbf{with}\ \gamma \subseteq \tau.\, \tau \rightarrow U \rightarrow \gamma \rightarrow \mathit{unit}$$

Here τ corresponds to the read class, *rc*, of the subject, *U* to the subject's write class, *wc*, and γ to the access class of the Ţle representing the multilevel input stream. This would permit the guard to be called with multilevel input streams. An exception would still be raised, though, if it were called by a subject whose read class is lower than some element in the stream, or whose write class is not unclassiŢed.

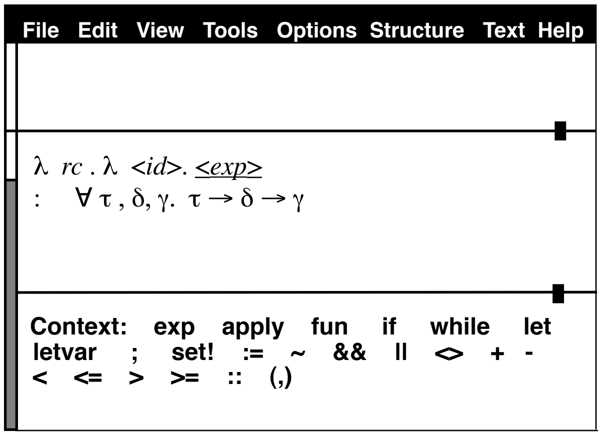


```
λ rc.  λ wc .
    let  i =   ropen (rc, INPUT_TS) in
    let o = wopen (wc, OUTPUT_S) in
    letvar v  := get (i)  in
    while  v  !=  EOF  do
      put(v, o);
      v  :=  get (i)
    end
: ?
```

```
Context:    exp e<exp> <exp> e fun  fix if while
let<id><exp>e let<id>e<exp> letvar<id><exp>e
letvar<id>e<exp> e;<exp> <exp>;e
```

Figure 7. Downgrade procedure

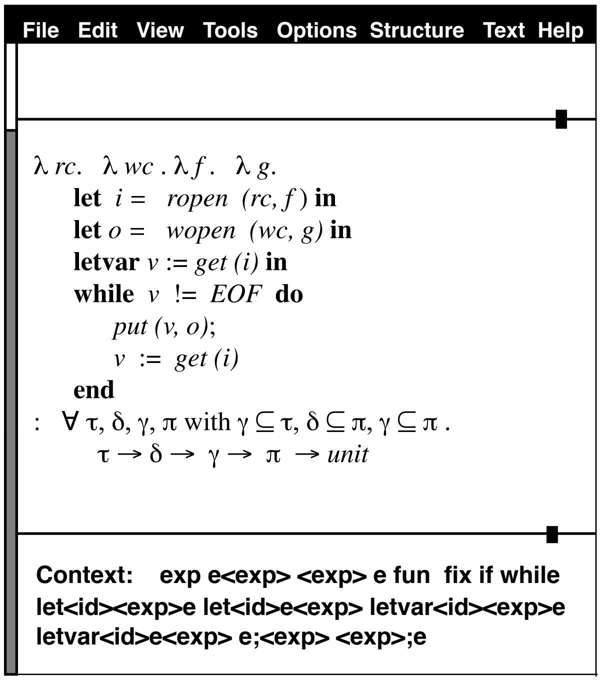
3.0.3 A downgrader: Sometimes the editor will detect a ţow policy violation which may be authorized. This is the



```
λ rc . λ <id>. <exp>
:   ∀ τ , δ, γ.  τ → δ → γ
```

```
Context:  exp  apply  fun  if  while  let
letvar  ;  set!  :=  ~  &&  ||  <>  +  -
<  <=  >  >=  ::  (,)
```

Figure 3. Editor window after supplying *rc* and selecting fun again



```
λ rc.  λ wc . λ f .  λ g.
    let i =  ropen (rc, f) in
    let o =  wopen (wc, g) in
    letvar v := get (i) in
    while v != EOF do
        put (v, o);
        v := get (i)
    end
:  ∀ τ, δ, γ, π with γ ⊆ τ, δ ⊆ π, γ ⊆ π .
       τ → δ →  γ →  π  → unit
```

```
Context:   exp e<exp> <exp> e fun  fix if while
let<id><exp>e let<id>e<exp> letvar<id><exp>e
letvar<id>e<exp> e;<exp> <exp>;e
```

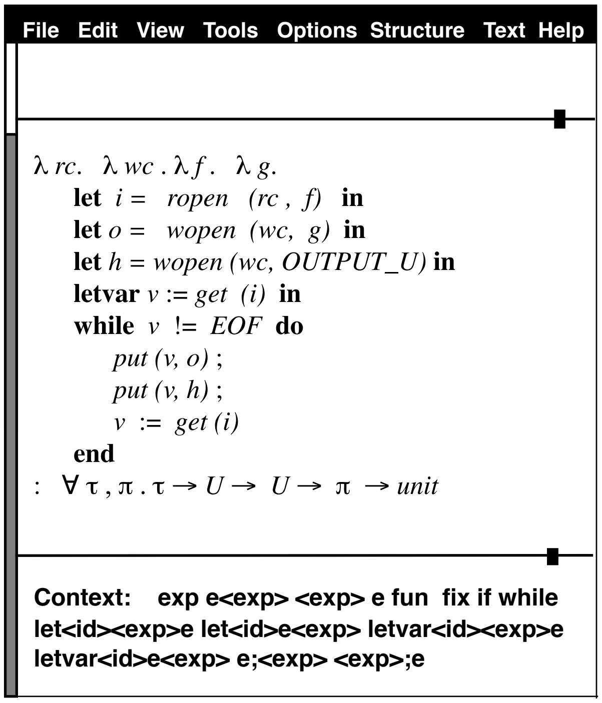
Figure 4. Editor window upon completion of secure copy procedure

Type variable τ is the type of *rc* and corresponds to the read class of the subject, δ is the type of *wc* and corresponds to the subject's write class, γ is the access class of *f*, and π the access class of *g*. The type indicates that the copy routine is capable of copying a Ţle *f*, classiŢed at level γ, to a Ţle *g* at an equal or higher level π ($\gamma \subseteq \pi$) as long as the read class of the subject dominates *f* ($\gamma \subseteq \tau$) and its write class is dominated by *g* ($\delta \subseteq \pi$), which is what we want. The editor allows a user to build a program while simultaneously inferring types at each step.

Now suppose we try to tamper with the procedure by also copying *f* to an unclassiŢed Ţle *OUTPUT_U*. We insert the declaration *wopen(wc, OUTPUT_U)* and the statement *put(v, h)* in the loop body. The result is shown in Figure 5. Notice what happened to the inferred type. It indicates that the procedure is not as general as one might have thought. SpeciŢcally, it can copy only unclassiŢed Ţles:

$$\forall \tau, \pi.\ \tau \rightarrow U \rightarrow U \rightarrow \pi \rightarrow unit$$

If invoked to copy any Ţle except an unclassiŢed one, it will raise an exception. In contrast, the type of the original Ţle-copy procedure indicates that it is able to copy Ţles classiŢed at any level. The fact that the type of the modiŢed procedure does not reţect this desired property is an indication that the procedure is suspicious.



```
λ rc.  λ wc . λ f .  λ g.
    let i =  ropen (rc , f)  in
    let o =  wopen (wc, g)  in
    let h = wopen (wc, OUTPUT_U) in
    letvar v := get (i)  in
    while v != EOF do
        put (v, o) ;
        put (v, h) ;
        v := get (i)
    end
:  ∀ τ , π . τ → U →  U →  π  → unit
```

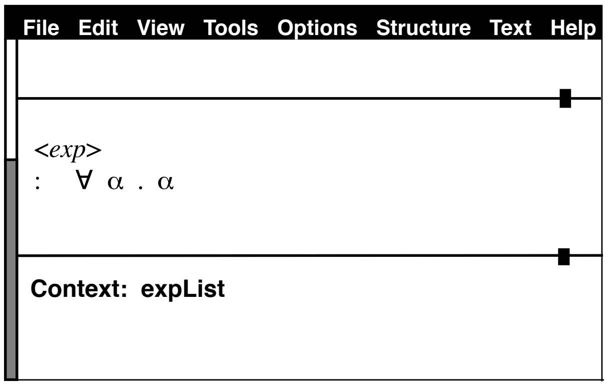
```
Context:   exp e<exp> <exp> e fun  fix if while
let<id><exp>e let<id>e<exp> letvar<id><exp>e
letvar<id>e<exp> e;<exp> <exp>;e
```

Figure 5. Rogue copy procedure

Consider what would happen if we tried to execute the rogue procedure without Ţrst subjecting it to our type (security) analysis. When called by a multilevel subject, say with write class unclassiŢed and read class secret, and a Ţle *f* at secret, the procedure will write a secret value to the unclassiŢed Ţle *OUTPUT_U*. Opening *OUTPUT_U* will succeed because the subject has write class of unclassiŢed. Unfortunately this is the state of affairs in practice today. Many programs are being used by multilevel subjects and they could very well be programs like this rogue procedure to another Ţle whose security class is at least as high. We will show how the editor we are developing can reveal an attempt to tamper with the procedure so that it also copies the input Ţle to an unclassiŢed third Ţle.

Our second example shows that there may be times when the editor will prohibit some code satisfying a certain speciŢcation from being written. The code must therefore be veriŢed within a more expressive logic. The editor can greatly reduce the amount of code that needs to be veriŢed in this way.

Finally, we illustrate the behavior of the editor when writing a procedure that downgrades information. In this case, the editor deems the function insecure. If, in fact, it is an authorized downgrade, then the function would have to be veriŢed in some other logic as above.



<exp>
: $\forall\ \alpha\ .\ \alpha$

Context: expList

Figure 1. Initial editor window

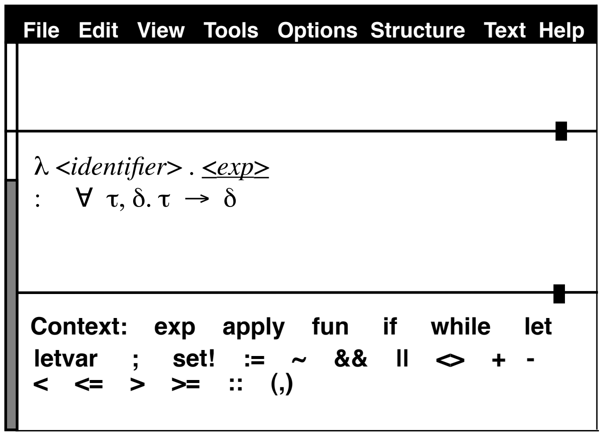
3.0.1 Secure file copy: The copy procedure has four parameters *rc*, *wc*, *f*, and *g*; *rc* and *wc* are the read/write classes of a multilevel subject, *f* is the Ţle to be copied and *g* is the result. Using the editor, we begin with the initial window in Figure 1. Here *<exp>* is a placeholder. All placeholders are delimited by angle brackets.

At each stage, the editor infers a type (security classiŢcation) for the program and displays it below the program. At this point, the editor reports that the type of the copy procedure is $\forall\ \alpha.\alpha$ because it knows nothing about its deŢnition as yet; it is only a placeholder thus far.

When the user selects *<exp>* with the mouse, a menu of choices for the various kinds of expressions appears at the bottom of the window. These choices permit the user to elaborate the expression. A sample of the selections and their results is given in Table 1. One of the choices, **fun**, is a function. Selecting it gives us a placeholder for a function of one argument, as shown in the window of Figure 2.

Table 1: Sample Editor Selections

Selection	Result	Type
fun	λ *<id>* *<exp>*	$\forall\alpha.\forall\beta.\alpha\rightarrow\beta$
while	**while** *<exp>* **do** *<exp>* **end**	*unit*
;	*<exp>* ; *<exp>*	$\forall\alpha\ .\ \alpha$
||	*<exp>* || *<exp>*	bool
let	**let** *<id>* = *<exp>* **in** *<exp>* **end**	$\forall\beta\ .\ \beta$
letvar	**letvar** *<id>* := *<exp>* **in** *<exp>* **end**	$\forall\gamma\ .\ \gamma$



λ *<identifier>* . *<exp>*
: $\forall\ \tau, \delta.\ \tau\ \rightarrow\ \delta$

Context: exp apply fun if while let letvar ; set! := ~ && || <> + - < <= > >= :: (,)

Figure 2. Editor window after choosing fun

The editor now infers a mapping type for the program but it still knows nothing as yet about the relationship between τ and δ. Next we Ţll in the identiŢer placeholder with *rc,* a subject's read class, and select another function for the expression placeholder, giving the window in Figure 3.

Now we get a type that looks more like the Ţnal type of the copy routine, but without more of the deŢnition available, nothing more can be inferred about its type. We now complete the deŢnition using operations from the TCB, giving the program in Figure 4.

The type inferred for the program now has three subtype constraints.

$$\gamma \subseteq \tau,\ \delta \subseteq \pi,\ \gamma \subseteq \pi$$

is actually an advanced type system, specifically, one supporting polymorphism and subtypes. There is a natural correspondence between information flow analysis and type checking. Ordinary types like `int` and `real`, for instance, can be replaced by security classes like `L` (low) and `H` (high). Further, just as `int` is a subtype of `real` in a traditional language, we can regard `L` as a subtype of `H`, reflecting the fact that information flow from `L` to `H` is permitted. A type system that supports polymorphism with respect to types therefore supports polymorphism with respect to security classes. Such a system affords us an opportunity to accurately capture information flow in procedures that accept inputs of arbitrary security classes, a source of difficulty in Denning's original approach.

We are developing an algorithm for deciding whether a given program has a type in our type system. This amounts to the algorithm having to find a *type derivation* for the program, using the rules of the type system. If a derivation cannot be found, then, since the rules characterize secure information flow , the algorithm reports that the program has an illicit flow with respect to a given flow policy . If a derivation can be found, then the algorithm reports that the program is secure by inferring a type for the program. The type reveals any flow assumptions, as subtype constraints, that are needed in the derivation.

We intend to implement the complete algorithm as a language-sensitive editor for a traditional block-structured language. This kind of editor is smart in that as a program is edited, it can be analyzed behind the scenes so that a programmer receives immediate feedback. The editor is said to be language sensitive because these analyses may determine whether a program satisfies certain restrictions of the language and, in some cases, even correct it if it fails to do so. We describe below a prototype language-sensitive editor that we are building for a subset of the full type system for secure information flow . It illustrates some important features of the algorithm we have developed thus far. We give editor snapshots that show how the editor responds to certain inputs during the course of writing some sample programs.

3. Sample Applications Using the Editor

Before showing how the editor behaves, we must first describe briefly some technical details of the type system on which the editor is based.

We take as our types the security classes *U, C, S,* and *TS*, ordered according to the following subtype relation:

$$U \sqsubseteq C \sqsubseteq S \sqsubseteq TS.$$

Intuitively, what this means is that *U* is a subtype of *TS* since flows from *U* to *TS* are permitted. These types form the primitive or base types of the system and may vary from one flow policy to another . There is a form of type in the system called a *type scheme* and it has the form

$$\forall\, \alpha_1 , \ldots , \alpha_n \textbf{ with } \kappa .\, \tau$$

The variables $\alpha_1,\ldots\ \alpha_n$ are called *type variables* and are universally quantified. For our purpose, they can be assumed to range over the primitive types. The set κ is a set of *subtype constraints*, expressed at the level of type variables and primitive types. For example, $\alpha \sqsubseteq U$ is a constraint that conveys α is a subtype of *U*. The symbol τ stands for a data type, which for our purpose, will consist only of a *mapping type* written $\tau_1 \rightarrow \tau_2$. An object of this type is a procedure that maps elements of type τ_1 to elements of type τ_2.

In our examples, we use operations from a hypothetical trusted computing base, or TCB. Among these are operations for opening files, for reading (*ropen*) and writing (*wopen*), and I/O (*get* and *put*). *Ropen* expects a subject's *read class (rc)* and a file to be opened and ensures that the read class dominates the access class of the file. On the other hand, given a subject's *write class (wc)* and a file to be opened, *wopen* ensures that the access class of the file dominates the write class.

We adopt *get* and *put* as our input and output operations; *get* (*i*) returns the next element of the input file descriptor *i*, and *put (v, o)* writes value *v* to the output file descriptor *o*. Each of these operations has a type built into the editor. For example, *put* expects to be called with a value classified at some level, say β, and a file descriptor classified at least as high, say δ, and writes the value to the file. This is conveyed by the type scheme

$$\forall\, \beta, \delta \textbf{ with } \beta \sqsubseteq \delta .\ \beta \rightarrow \delta \rightarrow \mathit{unit}$$

Here the type *unit* indicates that *put* is executed only for its effect and does not return a result. The subtype constraint ensures the *-property. Consequently, among the allowable types for *put* is

$$C \rightarrow S \rightarrow \mathit{unit}$$

since $C \sqsubseteq S$, but not

$$S \rightarrow C \rightarrow \mathit{unit}$$

because *S* is not a subtype of *C*. Notice that with *put* typed as above, we protect it against being called to write down for a multilevel subject.

We now give three sample programs to illustrate the editor. Each is intended to be executed by multilevel subjects. Our first example is a secure procedure to copy a file. It copies the elements of a file with some security class

A Practical Tool for Developing Trusted Applications

Cynthia E. Irvine Dennis Volpano
`irvine(volpano)@cs.nps.navy.mil`
Department of Computer Science, Naval Postgraduate School
Monterey, California 93943

Abstract

We introduce a tool we are developing that will allow designers of trusted applications to isolate those portions of a system where an information flow policy is being violated. The tool is a language-sensitive editor that checks a program for policy violations incrementally as the program is developed. What is novel about our approach is that the checking occurs as a form of type checking.

1. Introduction

Mandatory access control policies are concerned with the authorizations of individuals to access information based on its sensitivity. Within the context of automated information systems, each access by a subject to an object is mediated based on Ţxed labels associated with both. In general, applications will be executed by single-level subjects. If viewed from the perspective of the Bell and LaPadula model [2], a single-level subject will be constrained by the simple security property and the *-property such that it can neither obtain read access to objects which it does not dominate nor gain write access to objects which do not dominate it, respectively. Most applications, e.g. word processing systems, software engineering utilities, etc., can be executed by single-level subjects; in fact, careful application design can permit even complex multilevel data structures to be managed by single-level subjects [11].

There are, however, situations in which it is useful to have applications which are designed to violate the rules of a system's security policy enforcement mechanism, but which are trusted to do so only in a manner commensurate with externally-established authorizations. For example, in the case of the *-property of the Bell and LaPadula model [2], the application would be designed so that, when executed by a subject with a range of access classes, viz. a *multilevel subject,* the subject could read from an object with a high access class label and write to an object with a lower one. The multilevel subject will be constrained by the underlying mandatory policy, enforced through comparisons with the class deŢning the upper bound of its range on read accesses and by the lower bound of its range for write accesses.

Trusted applications are speciŢcally designed to be executed by multilevel subjects and are part of a system's trusted computing base (TCB). Careful security engineering of the combined trusted application and TCB will provide measurable assurance that the trusted application will behave as speciŢed and that its potential to violate security policy in an arbitrary and perhaps malicious manner is not realized. Unfortunately, current practice in the development of large systems does not always treat the engineering of the code to be executed by multilevel subjects with adequate rigor. Hence an environment ripe for the insertion and exploitation of malicious software exists. This makes certiŢcation, software maintenance, and continued accreditation difŢcult. This paper reports ongoing research to develop a new tool to support the development of large trusted applications.

Powerful formal systems for reasoning about security have been studied [3][10]. Given their expressiveness, and that often useful proof methodologies are missing, these systems have not been widely adopted in practice. We believe that this is because the methods offer, at most, formal systems to reason about security properties; typically there is no automated support for practitioners. However, reasoning about whether programs violate mandatory access control (MAC) policies, such as articulated in the Bell and LaPadula model [2], does not require such power. This was observed by Dorothy Denning in her seminal work on secure information ţow in computer systems. She described a way that compilers could efŢciently check programs for secure implicit and explicit information ţow [5], [6]. Unfortunately , this work was never widely used in practice, according to Denning [8].

2. An Editor for Trusted Applications

We are developing a new framework to carry out an extension of the analysis envisioned by Denning. The framework is a formal system of simple rules with which one can make judgements about information ţow in programs. What is interesting is that these rules follow rather directly from a *type system* for a *polymorphic* programming language with *subtypes*. Many such systems have been proposed [9][14][15].

So the technical basis for the tool described in this paper

Published in *Proceedings Eleventh Annual Computer Security Applications Conference*, New Orleans, LA, December 1995, IEEE Computer Society Press, Los Alamitos, CA, pp 190-195.